Choosing is Losing: How to combine the benefits of shallow and deep embeddings through reflection

ARTJOMS ŠINKAROVS, Heriot-Watt University, UK
JESPER COCKX, TU Delft, Netherlands

Dependently-typed host languages empower users to verify a wide range of properties of embedded languages and programs written in them. Designers of such embedded languages are faced with a difficult choice between using a shallow or a deep embedding. The former is easier to use because the entire infrastructure of the host langauge is immediately available. Meanwhile, the latter gives full access to the structure of embedded programs, but is difficult to use in practice, especially when the embedded language is itself dependently typed.

The main insight presented in this paper is that the choice between shallow and deep embedding can be eliminated by working in a host language with reflection capabilities: we start from a shallow embedding that can use all libraries and tools of the host language, and later use reflection to expose the deep structure of the embedded programs. Concretely, we apply this technique to embed three programming languages — Kaleidoscope, SaC, and (a subset of) APL — into the dependently typed theorem prover Agda, using dependent types to statically enforce several properties of interest. We then use Agda's reflection capabilities to extract the embedded programs back into the original language, so that the existing toolchain can be leveraged. In this process, statically verified properties of the host language are mapped onto runtime checks in the target language, allowing extracted programs to interact safely with existing code. Finally, we demonstrate the feasibility of our approach with the implementation and extraction of a convolutional neural network in our embedding of APL.

1 INTRODUCTION

Dependently-typed systems such as Coq [Bertot and Castran 2010], Agda [Norell 2008], Lean [de Moura et al. 2015] or Idris [Brady 2013] make it possible to write down specifications of programs as types and check that those specification are satisfied during type checking. These verified programs can be either executed directly, or compiled into a general-purpose target language. However, if we want to generate code targeting a specific tool or language that is not supported by the existing backends, we are stuck.

There are two common approaches to this problem: i) extending the toolchain to produce code in the desired target; or ii) defining a syntactic representation of the target language and implementing the program in it. The first approach gives the most freedom but it may be technically challenging, as one has to modify the internals of the theorem prover. The second approach forces us to define a deep embedding and express our specifications in it. While this seems easier up-front, the resulting encodings of the specifications get impractically large, especially when the embedded language is itself dependently typed.

In this paper we consider an approach that combines the benefits of the two aforementioned solutions. We use a shallow embedding of the target language and rely on *compile-time metaprogramming* to define custom extractors, making it possible to write a verified program hand-in-hand

Authors' addresses: Artjoms Šinkarovs, Heriot-Watt University, Edinburgh, Scotland, EH14 4AS, UK, a.sinkarovs@hw.ac.uk; Jesper Cockx, TU Delft, Delft, 2628 XE, Netherlands, j.g.h.cockx@tudelft.nl.

with an executor for it — all within a single environment. No need to touch internals of the host language; no encoding artefacts.

Specifically, we use *reflection* [Anand et al. 2018; Christiansen and Brady 2016; Ebner et al. 2017; van der Walt and Swierstra 2013] (also known as *direct reflection* [Barzilay 2005]), a form of compile-time metaprogramming.[1] That is, the host language has `quote`/`unquote` primitives that translate expressions to their internal representation and back. The internal representation is regular data that can be accessed and manipulated as usual. The main goal of reflection is to automate routine tasks of writing boilerplate code, *e.g.* tactics. However, APIs are typically also complete enough for general code generation tasks such as writing custom compiler backends.

In this paper, we are interested in writing verified embedded programs that can be extracted to an existing language. We approach this problem by using an embed-typecheck-extract loop. That is, we first embed a language or a subset of it into a theorem prover, using dependent types to encode properties of interest; next, we use the typechecker of the host language to verify the properties; and finally, we use a custom extractor to translate our implementation to the target language. Our approach is fine-grained: embeddings (and consequently extractors) do not have to cover the entire language, only the subset that is sufficient to specify the problem of interest. Also, instead of implementing the entire application in a theorem prover, one might chose a particular fragment and integrate it into an existing codebase.

Our proposed approach has several advantages:

- Firstly, working with a shallow embedding is much easier than working with a deep one. While deep embeddings provide full access to the program representation, they require us to encode the full syntax and typing rules of the target language, which soon becomes impractical for embedded languages with dependent types [McBride 2010]. This matters because we use dependent types in the embedding to encode the properties of interest. In contrast, shallow embeddings do not require any additional encoding, since they are regular programs in the host language, and can make use of all available libraries and tools.
- Secondly, specifying embeddings and extractions in the same dependently-typed framework means we can modify the extractor without rebuilding the whole compiler of the host language. Extractors can be fine-tuned for the particular application, *e.g.* treating a chosen function in a specific way. Extractors can also make full use of dependent types, ensuring that the extraction process is sound.
- Thirdly, since the embedded program and the extractor are developed side-by-side, we can make use of features of the host language to customise the behaviour of the extractor further. Concretely, we use Agda's *rewrite rules* as a lightweight method of applying verified domain-specific optimizations during extraction.

We implement extractors for Kaleidoskope [LLVM development team 2021] (Section 3) and SaC [Scholz 2003] (Section 4) in Agda. The former is a minimalist language that we use to demonstrate the basic extraction principles. The latter is a high-performance array language that generates efficient code for various architectures, but it has a restrictive type system. Our SaC embedding guarantees program termination, in-bound array indexing, and safety of arithmetic operations. Finally, we also embed a small subset of APL [Iverson 1962] into Agda (Section 5), which is sufficient to encode a Convolutional Neural Network [Šinkarovs et al. 2019]. Embedding APL is challenging because the language is untyped, and its basic operators are heavily overloaded. We define all the basic operators of APL in terms of the SaC embedding, effectively obtaining a compiler from APL into executable binaries.

---

[1]This use of the word 'reflection' is not to be confused with reflection in Java and similar languages, which is a form of *runtime metaprogramming*.

Contributions. The key contributions of this paper are:

- We introduce the concept of reflection-based extractors for shallowly-embedded languages that are embedded in metaprogramming-capable theorem provers.
- We extend the reflection API of Agda to make it better suited for defining custom extractors. Specifically, we extend the clause syntax with telescopes (Section 3.4), we add new operations for selective reduction (Section 3.5), and we implement optional reconstruction of datatype parameters (Section 4.2).
- We implement extractors for two languages: Kaleidoscope (Section 3) and SaC (Section 4). We use the latter as a basis for embedding a subset of APL (Section 5).
- We demonstrate how the proposed embeddings can be used by encoding a real-world application and ensuring that it extracts correctly (Section 5.2).

2 BACKGROUND

We start with a brief overview of key Agda constructions that are used in this paper. We also present relevant parts of the reflection API. For a more in-depth introduction to Agda refer to the Agda user manual [Agda development team 2021].

2.1 Agda Basics

Agda is an implementation of Martin-Löf's dependent type theory [Martin-Löf 1998] extended with many convenience constructions such as records, modules, do-notation, *etc.* Types are defined using the following syntax:

```
data ℕ : Set where          data Fin : ℕ → Set where          data _≡_ {a} {A : Set a}
  zero : ℕ                     zero : ∀ {n} → Fin (suc n)         (x : A) : A → Set a where
  suc : ℕ → ℕ                  suc : ∀ {n} → Fin n → Fin (suc n)    refl : x ≡ x
```

Unary natural numbers ℕ is a type with two constructors: zero and suc. Fin is an indexed type, where the index is of type ℕ. Constructor names can be overloaded and are disambiguated from the typing context, or can be prefixed with the type name: ℕ.zero, ℕ.suc. The Fin *n* type represents natural numbers that are bounded by *n*. In the definition of Fin, ∀ binds the variable without needing to specify its type. Curly braces indicate hidden arguments, which can be left out at function applications: we have suc zero : Fin 3, assuming that Agda can infer a (unique) value for the hidden argument. Hidden arguments can be passed explicitly using the syntax zero {*n* = *x*}. The propositional equality type _≡_ expresses equality of its two arguments, and has a single constructor refl stating that any value *x* is equal to itself. It uses mixfix syntax [Danielsson and Norell 2011]: the underscores in the name indicate placeholders for the arguments. Set is the name of the type of all small types. Sets form a predicative hierarchy, meaning that Set *i* is of type Set (ℓsuc *i*), and Set is a synonym for Set ℓzero. The functions ℓsuc and ℓzero are used to construct elements of type Level.

Functions are defined in a pattern-matching style:

```
_*_ : ℕ → ℕ → ℕ              abs : Fin zero → ℕ          wth : (a b : ℕ) → ℕ
zero    * y = zero            abs ()                     wth a b with a * b
(suc x) * y = y + (x * y)                                ... | zero = zero
                                                         ... | _ = b
```

The definition of abs uses the *absurd pattern* (), indicating an impossible case for the first argument, *i.e.* there is no constructor constructing a term of type Fin zero. Clauses with absurd patterns do not have a body, as the type system guarantees that they are never called at run-time. In the definition of wth we demonstrate the use of the with construction [McBride and McKinna 2004] which makes it possible to match on the result of an expression locally.

2.2 Reflection

Instead of explaining the full structure of all types that Agda uses to encode reflected syntax, we consider a small but representative sample: the function foo (left) and its reflection (right).

```
                        ‘foo = Definition.function
                          $ Clause.clause
                             []
                             (vArg (Pattern.con (quote ℕ.zero) [])) :: [])
foo : ℕ → ℕ              (Term.con (quote ℕ.zero) [])
foo 0        = zero    :: Clause.clause
foo (suc x) = x + x        (("x" , vArg (Term.def (quote ℕ) [])) :: [])
                           (vArg (Pattern.con (quote ℕ.suc) (vArg (Pattern.var 0) :: []))) :: [])
                           (def (quote _+_) (vArg (Term.var 0 []) :: vArg (Term.var 0 []) :: []))
                        :: []
```

The reflected function is defined by the list of clauses Clause.clause. Each clause has three arguments: i) the telescope, which is a list of free variables and their types; ii) the list of patterns; and iii) the body of the clause. The first clause does not have free variables, so the telescope is empty. The second clause has one variable called x. The pattern list in the first clause has one argument; vArg denotes that it is visible argument (hArg is used for hidden arguments). The actual pattern matches against the zero constructor, as expressed by Pattern.con, which has two arguments: the reflected constructor name and the list of reflected arguments. The number of reflected arguments must be the same as the number of the actual arguments, which is none in the case of zero. The quote primitive returns a representation of the name for the given Agda definition or constructor, which is of type Name. Variables (both in patterns and in terms) are given as de Bruijn indices into the telescope of the clause. That is, in the second clause the de Bruijn index 0 refers to the variable x. Note that we write 0 instead of zero, as numbers are expanded into their corresponding zero/suc representations.

Effectively using Agda's reflection API can be challenging because the syntax it uses matches the *internal* representation of Agda terms, which differs significantly from the surface syntax. Many constructs such as implicit arguments, instance arguments, let and with definitions exist only in the surface language. Translation from the surface language is performed by the *elaborator*. Following the approach of *elaborator reflection* introduced by Idris [Christiansen and Brady 2016], Agda exposes many parts of the elaborator to the reflection API, including reduction and normalisation of expressions. These operations are made available through the TC monad, which takes care of managing the current context of the elaborator.

The key metaprogramming primitives are quote and unquote, that operate as follows:

```
ex : (a : Bool) → if a then ℕ else Bool          macro getDef : Name → (Term → TC ⊤)
ex true = unquote helper                              getDef n h = do
  where helper : Term → TC ⊤                            d ← getDefinition n
        helper h = unify h (lit (nat 42))               t ← quoteTC d
ex false = false                                        unify h t
                                                  ‘foo = getDef foo
```

In ex, the unquote occurs in the true clause of the function. The argument to unquote is expected to be a function of type Term → TC ⊤, where ⊤ is the unit type. During elaboration, Agda creates a metavariable h of type ℕ, quotes it, and passes it to the function helper. In the body of helper, we call the TC operation unify h (lit (nat 42)) to unify the two expressions, instantiating h to the value

42. Finally, Agda replaces the expression unquote helper expression with the instantiated value of *h*. Overall, the effect of unquote helper is identical to just writing 42. However, the expression inside the helper can be arbitrarily complex and can depend on the syntactic structure of the term *h* as well as information obtained through operations in the TC monad.

Instead of quoting/unquoting explicitly, we can use the macro keyword to wrap any function with return type Term → TC ⊤. This takes care of quoting the arguments and unquoting the result. On the right, we define a macro getDef that obtains the definition of a name. The macro calls three functions from the reflection API. Firstly, getDefinition obtains the definition of the object with the name *n*. Secondly, quoteTC quotes the previously obtained definition (resulting in a doubly-quoted expression). Finally, we unify the quoted hole and the doubly quoted definition, so that after unquoting we get the reflected definition (and not the original one). We apply the macro in the last line, and as can be seen, no quote/unquote is needed. More details on reflection in Agda can be found in the user manual [Agda development team 2021].

3 BASIC EXTRACTION

In this section we demonstrate basic mechanisms that are necessary when implementing reflection-based extractors. To make our examples concrete, we use a minimalist language called Kaleidoscope [LLVM development team 2021] as our target. We explain the challenges and demonstrate Agda code snippets of the actual extractor for Kaleidoscope.

3.1 Framework Overview

The extraction process naturally splits into a language-dependent and a language-independent part. We start by explaining the reusable language-independent module, which we refer to the as the "framework". The entry point of the framework is a parametrised module Extract that contains a single macro called kompile:

```
module Extract (kompile-fun : Type → Term → Name → SKS Prog) where
  macro
    kompile : Name → Names → Names → (Term → TC ⊤)
    kompile n base skip hole = --
```

The *kompile-fun* parameter is a language-specific function specifying how to compile a single Agda function, given by its type, body and name. It operates within the SKS state monad, which it can use to register other functions that should also be extracted, and returns either a representation of the extracted function in the target language in case extraction succeeded, or an error message, as specified by the Prog type.

```
data Err {ℓ} (A : Set ℓ) : Set ℓ where      SKS : Set → Set  -- State Monad (KS)
  error : String → Err A                     TC : Set → Set   -- TypeChecking Monad
  ok : A → Err A                             Prog = Err String -- String or Error
```

Assuming kompile-fun is defined in the module Kaleid here is how to instantiate the framework:

```
open Extract (Kaleid.kompile-fun)
foo : ℕ → ℕ ; foo = ⋯ ; base-functions = ⋯ ; skip-functions = ⋯
extracted = kompile foo base-functions skip-functions
```

The first argument to the kompile macro is the name of the main function that we want to extract, in this case foo. The second and the third parameters of kompile are lists of names that control function inlining in the extracted terms and traversal into the definitions found in the call graph (which are added by the kompile-term function explained in Section 3.8). The kompile macro obtains the

normalised type and body of the main function, runs kompile-fun for the actual extraction and recursively extracts any functions that have been registered for extraction during the processing of foo.

To avoid repeated extraction of the same function, kompile keeps track of already compiled functions.[2] After all required functions have been compiled, the bodies of all extracted functions are concatenated and returned as the result of extraction.

3.2   Kaleidoscope

We borrow the Kaleidoscope example language from the tutorial on building frontends to LLVM [LLVM development team 2021]. Kaleidoscope is a minimalist first-order language with a single data type, the type of natural numbers[3], with basic arithmetic operations and comparisons. Following C convention, boolean values are encoded as numbers where 0 is false, and any other value is true. Function calls and conditionals operate as usual, and 'let' makes it possible to bind immutable values to variables. We extend Kaleidoscope with a one-argument assert operator that terminates the program if its argument evaluates to zero. Functions are defined by giving a name, a list of arguments and the expression for its body. External functions are defined by giving a name and a list of its arguments. We encode Kaleidoskope's AST as follows:

```
Id = String

data Op : Set where
  Plus Minus Times Divide : Op
  Eq Neq And Gt Lt : Op

data Expr : Set where
  Nat     : ℕ → Expr
  BinOp : Op → Expr → Expr → Expr
  Var     : String → Expr
  Call    : Id → List Expr → Expr
  Function : Id → List Id → Expr → Expr
  Extern : Id → List Id → Expr
  Let     : Id → Expr → Expr → Expr
  Assert : Expr → Expr
  If      : Expr → Expr → Expr → Expr
```

```
-- Recursive Fibonacci function:
fib = Function "fib" ("n" ∷ []) $
      If (BinOp Lt (Var "n") (Nat 2))
        (Nat 1)
        (BinOp Plus
          (Call "fib"
              [ BinOp Minus (Var "n") (Nat 2)])
          (Call "fib"
              [ BinOp Minus (Var "n") (Nat 1)]))
```

3.3   A shallow embedding of Kaleidoscope in Agda

To extract a Kaleidoscope program from an Agda program, we first need to identify what subset of Agda can be sensibly translated to Kaleidoscope. Let us start with the types. First, we need the natural number type ℕ as it is the main data type of Kaleidoscope. To describe invariants we also support the type Fin n of natural number strictly less than n, as well as the identity type _≡_ and the inequality type _<_ on natural numbers. The Fin type is mapped to numbers in the target language, while all proofs of _≡_ and _<_ are mapped to the constant 1. We also allow the decidability predicates Dec (a ≡ b) and Dec (a < b), which carry a boolean value and a proof that the relation holds or does not hold, depending on the value of the boolean. We map true to 1 and

---

[2]We had to postulate termination of this traversal as since the reflection API of Agda currently does not provide the guarantee that there is only a finite number of function symbols.
[3]The original version of Kaleidoscope used floats, but natural numbers are easier in the context of Agda as we can prove more properties of them.

false to 0, ignoring the proof. First order functions of the above types such as basic arithmetic _+_, _-_, *etc.* are mapped to corresponding functions in the target language.

While it is tempting to say that any Agda term of the above types could be translated into Kaleidoscope, this is not the case. For example, consider a function:

ex : ℕ → ℕ
ex *x* = length (showNat *x*)

where showNat returns a string representation of the given number. Neither length nor showNat are representable in Kaleidoscope, as there is no notion of strings in the language. To pin down precisely what fragment of Agda we can extract, we would have to restrict what types are allowed in embedded functions and what terms can appear in function bodies, taking us away from the world of shallow embeddings and into the world of deep embeddings. While it is certainly possible to define strongly typed deep embeddings in a dependently typed host language, all current solutions are very heavyweight when one has to deal with an embedded language that uses dependent types, as we do here (recall that we allow _<_, _≡_, *etc.*). In particular, one needs to encode not only types and terms of the embedded language, but also contexts and explicit substitutions, turning even the simplest programs into large and non-trivial terms.

It is still an open question whether there exists a satisfying middle ground between shallow and deep embedding. Our solution in this paper is to avoid the encoding problem entirely and rely instead on metaprogramming to extract a subset of Agda into our target language. An Agda term is defined to belong to the embedding if the extractor does not fail on it.

3.4 Normalisation

Working with a shallow embedding gives us an important benefit: we may use any host language constructs that are not present in the embedding, as long as they can be eliminated prior to extraction. For example, the target language may not support polymorphic or higher-order functions, yet we could write programs such as:

ex : (*n* : ℕ) → *n* < length "999" → ℕ
ex = ⋯

fib : (*k m n* : ℕ) → ℕ
fib 0   *m n* = *m*
fib (suc *k*) *m n* = let *m' , n'* = *n , m + n*
                     in fib *k m' n'*

In the type of ex, length is a function from String to ℕ, but it is applied to a constant string. In the second clause of fib we create a tuple (*n , m + n*) and immediately destruct it via pattern matching. Note that Kaleidoscope supports neither strings nor tuples, so neither length nor _,_ can be part of the final extracted Kaleidoscope code. However, if we simplify the terms, the result no longer contains any about strings or tuples, and hence can be extracted safely.

Such a simplification can be conveniently achieved by normalising the terms, *i.e.* by applying reduction rules to (sub)terms until they turn into values or neutral terms. Agda's reflection API offers a function normalise for this purpose. However, this only normalises the term itself and not the body of functions used in this term. This is a technical limitation that has to do with the internal representation of pattern-matching functions. To work around this limitation, we also recursively traverse the definition of each function used in the term and normalise all terms in their bodies.

During the implementation of this traversal, we were faced with the challenge of reconstructing the right typing context for each clause. Agda constructs this context internally during elaboration of the clauses, but the reflection API did not provide access to it. Rather than going through the painful and error-prone process of reconstructing this context, we instead extended the reflection API to provide it for us (see https://github.com/agda/agda/pull/4722 for the full story).

3.5 Controlling Reduction

Fully normalising a term sometimes leads to undesirable results. Consider the following program:

```
ex₅ : ℕ → ℕ
ex₅ x with x ≟ 42
... | yes _ = 10
... | no _ = 20
```

The definition of $\overset{?}{=}$ in the standard library is quite complex:

$$m \overset{?}{=} n = \text{map}' (\equiv^b \Longrightarrow \equiv m\ n)\ (\equiv \Longrightarrow \equiv^b m\ n)\ (\text{T?}\ (m \equiv^b n))$$

The four functions used in the body (*e.g.* map′, ≡ᵇ⟹≡, *etc.*) are not representable in Kaleidoscope, but comparison of natural numbers is. More generally, there is a common pattern where the host language represents a concept in a radically different way than in the target language. In such cases, we can customise the extractor by hard-coding the mapping of the Agda function to the target language function. For example, in this case we map $\overset{?}{=}$ to Eq.

To do this, we have to make sure that normalisation does not expand certain definitions. This is what the second argument (base) to our interface function kompile is used for — to specify the list of functions that must not reduce during normalisation. This functionality was not previously available in Agda, so we added two new primitives to the reflection API — dontReduceDefs and onlyReduceDefs — with pull request https://github.com/agda/agda/pull/4978. These functions give us an environment where any call to reduce or normalise will avoid reducing any function that is in the list (for dontReduceDefs) or not in the list (for onlyReduceDefs). This new feature is used by the kompile macro to avoid reducing functions for which we have a fixed translation.

3.6 Mapping Agda Types to Kaleidoscope Assertions

The next step after normalisation is to verify and translate the type signature of the embedded function into the target language. Kaleidoscope does not support type annotations, but we still need to verify that the function is first-order, and that the argument and return types are from the right universe. This is implemented in the kompile-ty function, of which we present a small fragment here:

```
record Assrt where
  constructor mk
  field v : Id ; a : Expr

_p+=a_ : PS → Assrt → PS ; _p+=a_ = ⋯

kompile-ty : Type → (pi-ok : Bool) → SPS (Err ⊤)
kompile-ty (def (quote ℕ) args) _ = return $ ok tt
kompile-ty (def (quote Fin) (arg _ x ∷ [])) _ = do
  ok p ← sps-kompile-term x where error x → ke x
  v     ← PS.cur <$> get
  modify $ _p+=a (mk v (BinOp Lt (Var v) p))
  return $ ok tt
kompile-ty _ _ = ⋯
```

It operates within the state monad SPS where the state is given by the type PS (pi-type state). As we traverse the type signature of a function, for non-dependent types such as ℕ we only verify whether the type is supported.

Dependent types such as Fin can be seen as encoding some predicate on their arguments as well as the element of the dependent type itself. We preserve this information by mapping them to assertions in the target language, essentially trading static checks for dynamic ones. Preserving this information is is important for two reasons. First, the toolchain can use this information for more aggressive optimisations. For example, the Kaleidoscope expression `if a > b: x else: y` can be simplified to `x` if we know from the types that `a` is greater than `b`. Second, extraction can be applied to partial programs. When extracted functions are being called externally, the calls are not typechecked, so the preconditions on the arguments that would normally be enforced by the type system may not hold. For example, given $f : (x : ℕ) → x > 0 → ℕ$, its first argument must not be zero. However, as $x > 0$ cannot be represented, external calls may pass zero to the extracted version of f.

In the case for Fin in the definition of kompile-ty, we first extract the argument x (obtaining a Kaleidoscope expression). Then we get the name of the function argument by referring to the PS.cur field of the state. Finally, we generate an assertion that checks whether the encoded witness is less than the argument to Fin (the upper bound), and add it to the state.

In general, to translate a dependent type $P : I → Set$, we start by picking basic types TI and TP that encode I and P, together with encoding functions $enc\text{-}i : I → TI$ and $enc\text{-}p : \{i\}(p : P\ i) → TP$. We then introduce a function $assrt\text{-}p : (ti : TI)(tp : TP) → Bool$ that decides whether the encoded arguments ti and tp come from a valid input, that is whether tp is the image under enc-p of some $p : P\ i$ where ti is the image of i. Finally, we prove soundness and completeness of the encoding:

sound-p : $∀ \{i\}\{p : P\ i\} → assrt\text{-}p\ (enc\text{-}i\ i)\ (enc\text{-}p\ p) ≡ true$
complete-p : $∀\ ti\ tp → assrt\text{-}p\ ti\ tp ≡ true → ∃_2\ λ\ i\ (p : P\ i) → ti ≡ enc\text{-}i\ i × tp ≡ enc\text{-}p\ p$

Dependent types with multiple arguments can be treated in exactly the same way.

Now, for decidable relations, we can entirely avoid encoding the proof object, as long as the computational behaviour of the function does not depend on the structure of the proof. This is in particular always the case for proof-irrelevant types such as _≡_ and _<_. We encode the elements of these types with the unit value (natural number 1). We then generate an assertion that uses the decision procedure. This decision procedure returns an element of the Dec type which we interpret as a boolean value: 1 for yes and 0 for no.

If a function returns a value of a dependent type, we also generate an assertion using the same rules. We organise the body of the extracted function such that there is a fixed variable binding the return value, and attach the assertion to that variable. Here is an example:

$ex_7 : (n : ℕ) → Fin\ (1 + n * n)$ `-- def ex7 (x_1):`
$ex_7\ n = fromℕ< \$\ n<1+n\ (n * n)$ `--   let n = x_1 ; __ret = n * n`
                               `--   let __ret_assrt = assert (__ret < 1 + x_1 * x_1)`
                               `--   __ret`

First, note the assertion for the return value, which has been generated from the return type of ex_7. Next, recall the types of fromℕ< and n<1+n:

fromℕ< : $∀ \{m\ n\} → m < n → Fin\ n$
n<1+n   : $∀\ n → n < 1 + n$

The fromℕ< function turns a proof of $m < n$ into an element of type Fin n. As we are encoding Fins as natural numbers, the extracted version of fromℕ< just returns the first argument of _<_,

which is the implicit argument *m*. Note that by doing so we are not losing any information, as the proof here is merely asserting that *m* fits the specification. In general, it is always possible to extract the type-level arguments of a dependent type such as _<_, as long as we avoid using features that distinguish between type-level and term-level arguments such as parametricity or run-time irrelevance.

It might seem that the assertion on the result is unnecessary, since it is guaranteed to be satisfied by construction. However, by inserting this assertion we pass on information (that might be undecidable to recompute) further down the toolchain. This may be used for example by the compiler of the target language to perform more optimizations. All these assertions can be turned off if a programmer or a compiler decides so, but this is not a concern of the extractor.

3.7 Pattern Matching

Many target languages do not support function definitions in a pattern-matching style, whereas in Agda it is the primary way of defining functions. Hence extractors often need to transform a definition by pattern matching into one using conditionals. In this section we show how to do this, and demonstrate how implementing the extractor in Agda can lead to extra safety guarantees.

The problem of compiling a definition by pattern matching splits naturally into two subproblems: computing a condition from each given clause, and joining all such conditions in a single conditional. Let us start with the latter. We implement the algorithm in the kompile-cls function of the following type:

kompile-cls : (*cls* : Clauses) → (*vars* : Strings) → (*ret* : String) → SKS (Err Expr)

The first argument is the list of clauses, the second argument is the list of variable names, and the last argument is the name of the variable we assign the return value to. As we are extracting the body of the clauses, we need to propagate the state of extraction, so the function operates in the SKS monad. The function traverses clauses in the order they appear in the definition and combines them in a nested if-then-else chain as in the following example:

```
ack : ℕ → ℕ → ℕ                          -- def ack (x1, x2):
ack 0 n            = 1 + n                --   if x1 == 0: 1 + x2
ack (suc m) 0      = ack m 1              --   else if x1 > 0 && x2 == 0: ack (x1-1) 1
ack (suc m) (suc n) = ack m (ack (suc m) n) --   else: ack (x1-1) (ack x1 (x2-1))
```

Note that in the second conditional, we have an explicit check that $x1 > 0$, which is redundant. However, if the first and the second clauses were swapped, such a comparison with zero must be present. Our current implementation takes a minimalist approach and generates predicates for each clause separately, without taking the previous clauses into account. Many target languages will optimize away such redundant checks.

To compile definitions with absurd clauses, we need a way to abort computation. For example:

```
ex₁₁ : ∀ n → n < 0 → ℕ -- def ex11 (x1, x2):
ex₁₁ n ()                --   assert (0)
```

Rather than returning an arbitrary value, we use assert (0) to abort computation. When a function has both regular and absurd clauses, we are faced with a design decision. We can either preserve the absurd clauses and use assert (0) as their body, or eliminate them entirely. While eliminating them is sound, leaving them in can provide valuable information for the compiler of the target language. Consider the following example:

```
ex₁₀ : let P x = x * x + 2 ∸ 3 * x in ∀ x → 0 < P x → ℕ
ex₁₀ 1 () ; ex₁₀ 2 () ; ex₁₀ x pf = ⋯
```

We can generate an assertion that P x is greater than 0, and after eliminating first two cases, the body of the function would reduce to a single statement over variables x and pf. However, deducing that x does not equal 1 or 2 is not straightforward. Instead, we preserve this information as an assertion, so the compiler of the target language can make use of it.

The actual translation from patterns to predicates is implemented in kompile-clpats. Here we show only two clauses for compiling a function that matches on the Fin constructors zero and suc, respectively.

```
{-# TERMINATING #-}
kompile-clpats : Telescope → (pats : List (Arg Pattern)) → (exprs : List Expr) → PatSt → Err PatSt
kompile-clpats tel (arg i (con (quote Fin.zero) ps) :: l) (e :: ctx) pst =
  kompile-clpats tel l ctx $ pst +=c (BinOp Eq e (Nat 0))
kompile-clpats tel (arg i (con (quote Fin.suc) ps@(_ :: _ :: [])) :: l) (e :: ctx) pst = do
  (ub , pst) ← pst-fresh pst "ub_"
  kompile-clpats tel (ps ++ l) (Var ub :: (BinOp Minus e (Nat 1)) :: ctx)
                  $ pst +=c (BinOp Gt e (Nat 0))
kompile-clpats tel ps ctx pst = ···
```

The function take four arguments: the telescope mapping variables used in the pattern list to their names and types; the list of patterns; the list of expressions that are being matched by the patterns; and the state record PatSt. When compiling a clause, the list of expressions is initialised with the function arguments. The state record contains the counter for fresh variables, the list of conditions accumulated so far, local assignments, and so on.

For each constructor pattern we produce a condition that holds only when the encoded value in the target language represents the value that was built using the given constructor. For example, as we represent Fin with natural numbers, the conditions for matching e against a pattern zero $\{ub\}$ constructor is $e == 0$ (note that the precondition generated from the type of e already enforces that $e < ub$, so we do not need to check it again). Correspondingly, the pattern suc $\{ub\}$ x yields the condition $e > 0$. The pst-fresh function generates a variable with the unique (within the clause) name, and _+=c_ adds the new condition to the state.

Note that we marked this function as terminating. We had to do so as the recursive call to kompile-clpats with argument $ps ++ l$ is not structurally decreasing. Nevertheless, the function is terminating because the Pattern type is well-founded, and all the objects of that type are finite. The full version of our code works around this limitation of the termination checker by adding an extra argument expressing the overall size of the pattern, but we omit it here for simplicity.

As a side note, Agda internally represents definitions by pattern matching as *case trees* where most redundant checks have been eliminated. However, unfortunately this representation is currently not available via the reflection API. However, if we wanted to generate more efficient code it would probably be simpler to extend the reflection API rather than reimplement the translation to a case tree ourselves.

3.8 Translating terms

The actual translation of Agda terms into Kaleidoscope terms is a mechanical process. However, as the translation may fail, the use of monads and do-notation for managing errors help us to keep the code clean:

```
kompile-term : Term → Telescope → SKS (Err Expr)
kompile-term (def (quote _+_) args@(arg _ a :: arg _ b :: [])) vars = do
  a ← kompile-term a vars
```

```
    b ← kompile-term b vars
    return $ BinOp <$> ok Plus ⊛ a ⊛ b
kompile-term (def (quote F.fromℕ<) args) vars = do
    ok (x ∷ []) ← kompile-arglist args (0 ∷ []) vars
                where _ → kt "wrong assumptions about arguments of fromℕ<"
    return $ ok x
kompile-term (def n args@(_ ∷ _)) vars = do
    modify λ k → record k { funs = KS.funs k ++ [ n ] }
    args ← kompile-arglist args (mk-iota-mask $ length args) vars
    return $ Call <$> ok (normalise-name $ showName n) ⊛ args
kompile-term t vars = ⋯
```

We demonstrate three representative clauses of the term extracting function. First, we turn SKS and Err into monads by defining their bind and return actions. As each monad is an applicative functor, we get _<$>_ and _⊛_ operations for free. The instance resolution mechanism[4] makes it possible to overload monadic/functorial operations without explicitly mentioning in which monad we are operating.

A typical case of kompile-term extracts the arguments and puts them together in a corresponding expression of the target language, as for example in the case for _+_.

For some constructions it is convenient to handle arguments without explicitly pattern-matching them, *e.g.* some constructor with a long argument list where we are interested only in a particular subset. For such reasons we introduce the kompile-arglist function where the first argument is the list of Arguments; and the second argument is the mask that specifies indices into the first argument. The function extracts each argument from the list as specified by the mask. In case of fromℕ< we use this function to extract the first argument from the *args* list.

The last clause deals with general function calls that do not require special treatment. We first ensure that argument list is non-empty: the _∷_ pattern. Then we add the name of the function into the funs field of the state record, which is used by kompile to extract all the necessary dependencies. Then we extract the arguments, using the helper function mk-iota-mask that generates indices from 0 to the length of the argument list. Finally we create an Expression for a function call. We use the extracted arguments and we normalise the name to get rid of unicode symbols.

3.9 Example

Let us consider the actual output that our extractor generates for a reasonably complex function: the binary logarithm. We take the simplest specification, and we assume that logarithm of zero is zero. One difficulty with this function is that it is not structurally recursive and Agda does not recognise that it terminates. We use a standard technique of recursing on a well-founded _<_ predicate (inequality of natural numbers) to prove termination. Here is the Agda definition and the extracted code (slightly reformatted) arranged side-by-side:

```
x<m⟹sx/2<m : ∀ x m → x < m → suc x / 2 < m
x<m⟹sx/2<m x m x<m = ≤-trans (m/n<m (suc x) 2 (s≤s z≤n) ≤-refl) x<m
-- Extracted with command : kompile log₂ (quote ≤-refl ∷ quote _<_ ∷ []) []
log₂′ : ∀ {m} → (n : ℕ) → (n < m) → ℕ    -- def log2' (x_1, x_2, x_3):
                                          --    let x_3_assrt = assert (x_2 < x_1)
log₂′ {m} 0 _ = 0                         --    let __ret = if (x_2 == 0):
```

---

[4]For the details see the "Instance Arguments" section of Agda's manual [Agda development team 2021].

```
                                    --      let m = x_1 ; x = x_3
                                    --        0
log₂′ {m} 1 _ = 0                   --      else if (x_2 > 0) && (x_2 - 1 == 0):
                                    --        let m = x_1 ; x = x_3
                                    --        0
log₂′ {suc m} n@(suc x) n<m =       --      else if (x_1 > 0) && (x_2 > 0):
  1 + log₂′ {m = m} (n / 2)         --        let m = x_1 - 1 ; x = x_2 - 1 ; n<m = x_3
    (x<m⟹sx/2<m x m $ ≤-pred n<m) --        1 + log2' (m, 0 + (x + 1 - 0) / (1 + 1), 1)
                                    --      else:
                                    --        assert (0)

log₂ : ℕ → ℕ                        -- def log2 (x_1):
log₂ x = log₂′ x ≤-refl             --    let x = x_1 ; __ret = log2' (1 + x, x, 1)
                                    --    __ret
```

We define two functions: a wrapper function log₂, and a helper log₂′ where the actual work happens. We define two base cases for 0 and 1 and the recursive case on the argument divided by 2. Note that the function $\_/\_$: $(x\ y$: $\mathbb{N})\{\neq 0$: $\mathsf{False}(y \stackrel{?}{=} 0)\} \rightarrow \mathbb{N}$ takes an implicit argument that asserts that the divisor is non-zero. Agda is capable to deduce the proof automatically for constant values such as 2. In the extracted code we start with the assertion that was extracted from the type of $n<m$. The first case is trivial. In the second case we see $x\_2 - 1 == 0$, rather than $x\_2 == 1$, which is an artefact of the algorithm used in kompile-clpats and the fact that 1 is represented as suc zero. This is a correct translation, as we ensure that x_2 is greater than zero before subtracting one. However, this could be further optimised either by the target language or as a post-extraction step.

In the recursive case, division looks suspiciously complex. The reason for the complexity of the division operation is because $\_/\_$ in Agda is defined in terms of a helper function div-helper $k\ m\ n\ j$ that corresponds to the expression $k + ((n+m-j)\ /\ (1+m))$. We could have suppressed the normalisation of $\_/\_$, but this is not generally desirable, as it prevents evaluation prior to extraction. For example, without suppression $(2+n)/2$ normalises to $1+(n/2)$, whereas with $\_/\_$ suppressed it would be treated as an opaque object.

Also note how the recursive call uses the value 1 instead of an actual proof. Per our assumption, $\_<\_$ is used as a static assertion (we cannot pattern-match on its value). This means that any function that has a return type $a < b$ can be replaced with the unit value. This is valid, because we are extracting function calls that were verified by the typechecker. Therefore, we can completely omit the proof part of $\_<\_$, only acknowledging that the type is inhabited. This particular case relies on ≤-trans (from the inlined proof) and ≤-refl (from the log₂ definition) being extracted into unit values.

There is also the final else case, which is not specified in the original code on the left. The reason for this extra case is that our pattern matching is not complete. We are missing the case where m is zero and n is greater than 2. While Agda's coverage checker agrees that such case is impossible, it automatically inserts the missing case into internal representation as an absurd clause.

3.10 Rewriting

A common way to define domain-specific compiler optimizations is through the specification of *rewrite rules* that rewrite terms matching a given pattern to an equivalent form that is either more efficient or reveals further optimization opportunities. By giving a shallow embedding of our target

language in Agda, we have the opportunity to define *verified* rewrite rules, providing a proof that the left- and right-hand side of the rewrite rule are equivalent. To achieve this, we could define our own representation of verified rewrite rules and integrate them into the extractor. However, we can avoid the effort of doing so since Agda already has a built-in concept of rewrite rules.

Rewrite rules were originally introduced to Agda to work around the limitations of definitional equality in intentional type theory. For example, it can be used to make $0 + x$ definitionally equal to $x + 0$. Since we work with a shallow embedding, these rewrite rules are equally well suited to optimize the embedded programs we write before they are extracted. A typical example of a rewrite rule in Agda rewrites expressions of the form $x + 0$ to x:

```
plus-0 : ∀ x → x + 0 ≡ x
plus-0 zero    = refl
plus-0 (suc x) = cong suc $ plus-0 x
{-# REWRITE plus-0 #-}
```

The definition of plus-0 proves the equivalence of the left- and right-hand sides of the rule, and the REWRITE pragma registers it as a rewrite to be applied automatically during normalisation. As another example, we define the following rule that we were taught in school:

```
open import Data.Nat.Solver ; open +-*-Solver
sum-square : ∀ x y → x * x + 2 * x * y + y * y ≡ (x + y) * (x + y)
sum-square = solve 2 (λ x y → x :* x :+ con 2 :* x :* y :+ y :* y := (x :+ y) :* (x :+ y)) refl
{-# REWRITE sum-square #-}
```

It might seem that such an example of rewriting expressions over natural numbers are not very practical, but the benefit becomes more obvious for more complex data structures. For example, here is the famous fusion law for distributing maps over function composition _∘_:

```
map-∘ : ∀ {X Y Z : Set}{g : X → Y}{f : Y → Z} → ∀ xs → (map f ∘ map g) xs ≡ map (f ∘ g) xs
map-∘ []       = refl
map-∘ (x ∷ xs) = cong (_ ∷_) (map-∘ xs)
{-# REWRITE map-∘ #-}
```

Instead of traversing all the elements of xs and then all the elements of the map g xs, we can compute the same result in a single traversal. Generally, we often know a number of properties about the data structures we are working with. In a dependently-typed systems we can make those properties explicit and formally verify them, but in rewriting-capable systems we can selectively turn properties into optimisations.

One danger with rewrite rules is that we can get different result depending on the order of rule application. The recently introduced confluence checker [Cockx et al. 2021] helps to prevent this problem. When it is turned on, it reports when the registered set of rewrite rules is not confluent. For example, in case of plus-0 rule, the confluence checker complains that plus (suc x) zero can be rewritten to either (suc x) or suc (plus x zero). If we add a new rewrite rule for plus (suc x) zero ↦ suc x, our rewrite system is again accepted.

3.11 Monadic Workaround for Lets

One of the unfortunate design choices of the Agda internal language is the lack of a 'let' construct. All the lets we use in the code are eliminated eagerly through substitution of the bound expression in the body. While this is semantically sound, it leads to unnecessary code duplication:

$ex_8 : \mathbb{N} \to \mathbb{N}$
$ex_8\ x = $ let $a = x \ {}^* \ x + 3 \ {}^* \ x + 5$ in $a + a\ \text{-\,-} \ \Rightarrow\ (x\ {}^*\ x\ +\ 3\ {}^*\ x\ +\ 5)\ +\ (x\ {}^*\ x\ +\ 3\ {}^*\ x\ +\ 5)$

While changing Agda itself to support 'let' in the internal language would be a major change, we can use the following elegant workaround. Agda's do-notation is a syntactic sugar that expands to the monadic bind `_»=_`. In particular, we can work in the identity monad by defining $a\ \text{»=}\ f = f\ a$ and adding it to our extractor, allowing us to use do-notation instead of let:

$ex_8' : \mathbb{N} \to \mathbb{N}$
$ex_8'\ x = $ do $a \leftarrow x\ {}^*\ x + 3\ {}^*\ x + 5;\ a + a$

4 ARRAY LANGUAGE

In this section we switch to extraction of a different language called Single Assignment C — SAC for short. We explain essence of the language, our embedding for it in Agda, and the difference in the extraction process when comparing to Kaleidoscope.

4.1 SAC — Single Assignment C

SAC is a first-order array language that looks like C syntactically, but nonetheless is purely functional. The main goal of SAC is to provide a framework to efficiently operate with multi-dimensional arrays. All types in SAC represent arrays with potentially unknown ranks (number of dimensions) and shapes (extents along dimensions). Its purely functional nature is achieved by ruling out expressions that have side effects, undefined behaviour, pointers, and other imperative constructions of C. This allows the compiler to use implicit memory management and to make decisions about parallel execution of certain code parts without requiring any explicit annotations. The current compiler sac2c supports efficient compilation to multicore and GPU architectures [Šinkarovs et al. 2014]. We introduce the key aspects of the language that are used in the extraction examples. For more information about SAC refer to [Grelck and Scholz 2006].

Type system. The main distinctive features of SAC are its hierarchy of array types, intersection types and the unified data-parallel array comprehensions. In SAC, functions express rank-polymorphic computations. That is, they compute on arrays of arbitrary rank and shape. The type system tracks information about shapes by using explicit attributes. For example, arrays where elements are integers and the shape is statically known are expressed as:

```
int[] scal;        int[42] vec;        int[10,10,10] ten;
```

The shape of an array at runtime is always given by a tuple of natural numbers. In types, the shape attribute is an approximation of the runtime value. In the example above array scal is of rank zero, representing a scalar value. The vec is a 1-dimensional array containing 42 elements, and ten is a 3-dimensional array of shape $10 \times 10 \times 10$.

Arrays with static dimensions but without a static size can be specified using a dot:

```
int[.] a;        int[.,.] b;        int[.,.,.] c;
```

The variables a, b and c are of ranks one, two, and three respectively, with unknown shapes. Finally, arrays can have a dynamic number of dimensions:

```
int[+] d;        int[*] e;
```

where d is an array of rank 1 or higher, and e is an array of any rank.

There is a natural partial order on type attributes according to the precision with which they describe the rank and dimensions of an array:

```
[] ≤ [*]        [42] ≤ [.] ≤ [+] ≤ [*]        [2,2] ≤ [.,.] ≤ [+] ≤ [*]
```

This shape hierarchy gives rise to function overloading based on the shape of the arguments, where the compiler picks the most specific instance in case of overlap.

A limitation of SaC is that there is no way to express complex shape relations in case of statically unknown shapes. For example, it would be useful to specify matrix multiplication as:

```
int[m,n] matmul (int[m,k], int[k,n])
```

Unfortunately, this cannot be expressed as there is no notion of type-level variables. This becomes even more problematic for functions like take or drop below:

```
int[.] take(int n, int[.] v)  // take(2, [1,2,3,4]) == [1,2]
int[.] drop(int n, int[.] v)  // drop(2, [1,2,3,4]) == [3,4]
```

Annotating them with a precise size would require some form of dependent types, which means that we would have to give up global type inference.

The key language construct in SaC is the with-loop — a data-parallel array comprehension construct. The programmer specifies how index sets are to be mapped into element values and whether the computed values form an array or are folded into a single value. This could be also thought of as a generalised map/reduce construct. Consider an example of matrix multiplication:

```
int[.,.] matmul (int[.,.] a, int[.,.] b) {
  M = shape(a)[0]; K = shape(a)[1]; N = shape(b)[1];
  return with {
    ([0,0] <= [i,j] < [M,N]): with {
        ([0] <= [k] < [K]): a[[i,k]]*b[[k,j]];
    }: fold (+, 0);
  }: genarray ([M,N], 0);
}
```

First, we obtain the number of rows and columns in the matrices by querying their shape (which is a 1-dimensional array) and selecting its corresponding components. The outer with-loop specifies the index range from $[0, 0]$ up to $[M, N]$ and tells that all the computed values should be put into the array of shape $M \times N$. The latter is specified with genarray at the end of the with-loop, where the first argument is the shape of the result, and the second one is the default element. The default element serves two purposes: i) providing a value for the array indices that were not specified in the index ranges; ii) providing the shape of each element. The latter is important because the computed elements must all have the same shape. The shape of the result is a concatenation of the genarray shape and the shape of the default element. The inner with-loop computes the sum of point-wise multiplied i-th row and j-th column, expressed by fold (+, 0). For more details on programming in SaC refer to [Grelck 2012].

4.2 Embedded Array Language

To embed SaC, we have to define a type of multi-dimensional arrays, and three constructs: with-loops, shapes, and selections. Our goal is to express non-trivial shape relations between the arguments of a function and to ensure in-bound array indexing statically. We achieve this with the following two Agda types:

```
data Ix : (d : ℕ) → (s : Vec ℕ d) → Set where
  []   : Ix 0 []
  _∷_  : ∀ {d s x} → Fin x → (ix : Ix d s) → Ix (suc d) (x ∷ s)
```

```
record Ar {a} (X : Set a) (d : ℕ) (s : Vec ℕ d) : Set a where
  constructor imap
  field sel : Ix d s → X
```

Both types are indexed by a shape s, represented as a Vector of natural numbers. The Ix type is a type of valid indices within the index-space generated by the shape s. A valid index in such an index-space is a tuple of natural numbers that is component-wise less than the shape s. Finally, the array with elements of type X is given by a function from valid indices to X. In some sense Ar and Ix are second-order versions of Vec and Fin. This could be also thought of as a computational interpretation of the Mathematics of Arrays [Mullin 1988] (where Ψ becomes an array constructor), or as a generalisation of pull arrays [Svensson and Svenningsson 2014], or as simple containers [Altenkirch et al. 2015].

This encoding intrinsically guarantees that all the array accesses are within bounds. As for `fold` with-loops, there is no need for a special construct: we can define a recursive function (analogous to reduce on Vec), and let the extractor translate its applications into the corresponding `fold` with-loop.

Consider now the matrix multiplication example expressed in the embedded language:

```
mm : ∀ {a b c} → let Mat x y = Ar ℕ 2 (x ∷ y ∷ []) in Mat a b → Mat b c → Mat a c
mm (imap a) (imap b) = imap body where
    body : _
    body (i ∷ j ∷ []) = sum $ imap λ where (k ∷ []) → a (i ∷ k ∷ []) * b (k ∷ j ∷ [])
```

With a similar level of expressiveness, the implementation encodes the correct shape relation between the arguments and guarantees in-bound indexing without any explicit proofs.

Our definition of Ar satisfies the useful property that any composition of operations on arrays normalises to a single imap. Consider an example:

```
_*_ _+_ : ∀ {d s} → (a b : Ar ℕ d s) → Ar ℕ d s
_+_ a b = imap λ iv → sel a iv ℕ.+ sel b iv
_*_ a b = imap λ iv → sel a iv ℕ.* sel b iv

_ᵀ : ∀ {X : Set}{d s} → Ar X d s → Ar X d (reverse s)
_ᵀ a = imap λ iv → sel a (subst (Ix _) (reverse-inv _) (ix-reverse iv))

ex : ∀ {m n} → Ar ℕ 2 (n ∷ m ∷ []) → Ar ℕ 2 (m ∷ n ∷ []) → Ar ℕ 2 (m ∷ n ∷ [])
ex a b = aᵀ + (b * b)
```

Here we defined `_+_` and `_*_` as element-wise operations on the array elements. The `_ᵀ` is a transposition of the matrix, which reverses the order of the dimensions. Note that all of these are defined in a rank-polymorphic style. For transposition we had to apply a proof (reverse-inv) that reversing an index is involutive. The body of ex is given as four operations on the entire arrays, conceptually creating a new copy of an array at every application. Due to our encoding, the body of ex normalises into a single imap. This is largely possible because we defined Ar as a record, and these are guaranteed to preserve η-equality. That is, every x : Ar d s is *definitionally* equal to imap (sel x).

During the implementation of our extractor for SaC in Agda, we encountered an unexpected challenge related to the definition of Ar. By defining it as a record type, the elaborator of Agda decides to erase the (implicit) arguments a, X, d, and s of the constructor imap, replacing them by the constructor unknown in the reflected syntax. The reason why Agda does this is because these

parameters can always be reconstructed from the type of the array. However, inferring them is far from trivial as imap may appear in arbitrary contexts. To work around this issue, we extended the reflection API of Agda with a new primitive withReconstructed that instructs all the further calls to getDefinition, normalise, reduce, *etc.* to reconstruct the parameters that are normally marked as unknown. We use this function when kompile obtains the representation of each definition. For more details on this new feature, see https://github.com/agda/agda/pull/5022.

4.3 Validating Types

One of the major differences between extracting into Kaleidoscope and into SaC is the presence of the non-trivial type system in the latter. This requires us to choose what Agda types are going to be supported and how to translate them into the target language.

SaC lacks support for heterogeneously nested arrays: all the elements in the array must be of the same shape. Therefore, there is no way to construct the following types:

```
(int[.])[5]        (int[.])[.]        (int[*])[.]        (int[*])[*]
```

Furthermore, syntactically, there is no way to express a nested array type. However, one can deal with homogeneous nesting by flattening as follows:

```
(int[5])[6] => int[6,5]            (int[τ])[σ] => int([σ] ++ [τ])
```

Also, the with-loop construct makes it possible to express the computation in a nested style, but the resulting array type is flattened according to the scheme above. Consider an example:

```
int[5] foo (int[1]);  // some function that produces 5-element vectors.
int[6] gen (int[6,5] a) {
  return with {
    ([0] <= iv < [6]): foo (iv);
  }: genarray ([6], with{}: genarray ([5], 0));
}
```

The function gen computes a two-dimensional array. The with-loop that this array generates has a 1-dimensional index-space (specified by the shape [6]), and non-scalar elements. The latter is given by the shape of the default element, which is a vector of 5 zeroes. As a result we get an array of shape [6,5].

This suggests that nested imaps can be mapped directly to with-loops, translating nested array types in Agda into flattened types in SaC. However, while imap is a constructor for Ar, there is also the projection sel. Selecting into a nested array would result in selection on a partial index:

partial-sel : Ar (Ar ℕ 1 (5 ∷ [])) 1 (6 ∷ []) → Ar ℕ 1 (5 ∷ [])
partial-sel x = sel x (zero ∷ [])
-- int[5] partial_sel (int[5,6] a) { return ?? }

The argument to partial-sel is a nested array of shape [6], with inner elements of shape [5]. We can represent it in SaC as an array of shape [5,6]. Selections into such an array require two-element indices, but in the above code, selection happens on a 1-element index. Fortunately, we can generalise SaC selections as follows:

```
int[*] sel(int[.] idx, int[*] a) {
  sh_inner = drop (shape (idx), shape (a));
  return with {
    (0*sh_inner <= iv < sh_inner): a[idx ++ iv];
```

```
    }: genarray (sh_inner, 0);
  }
```

When selecting an array a at index idx, the shape of the result is computed by dropping idx-many elements from the shape of the argument. The content of the result is given by a mapping iv ↦ a[idx ++ iv], where iv iterates over the index space of the resulting array. Essentially, we partially apply the selection operation to idx. Partial selection is a well-known pattern and it is defined in the standard library for all the supported base types such as int, float, *etc*.

Array shapes in Agda are represented by the Vec type, whereas SaC shapes are 1-dimensional arrays. Mapping a vector type is straight-forward, as we only need to implement nil/cons to construct vectors and head/tail to eliminate them[5]:

```
int[.] cons (int x, int[.] xs) {          int[.] tl (int[.] xs) {
  return with {                             return with {
    ([0] <= iv <= [0]): x;                    (. <= iv <= .): xs[iv+1];
    ([1] <= iv <= .): xs[iv - 1];           }: genarray (shape (xs) - 1); }
  }: genarray (1 + shape (xs));
}                                         int hd (int[.] xs) { return xs[0]; }
```

Finally, note that if we can extract Vecs, we can extract Lists into exactly the same target language constructs. The only difference lies in the analysis of the nesting of the type. Ar of Vec and Vec of Ar are always homogeneous as long as the leaf element is some base type like ℕ. Lists of base types or lists of homogeneous arrays are also homogeneous. However, whenever List shows up on any inner level of the nesting, we loose homogeneity, *e.g.* List∘List is inhomogeneous, because the inner elements may be of different sizes. We implement this analysis in the extractor, therefore allowing for the combination of nested Ar, Vec and List over the base types ℕ and Float.

5 APL AND CNN

In this section we consider the embedding of an APL subset that is large enough to port the implementation of a convolutional neural network [Šinkarovs et al. 2019]. APL presents an interesting case for our approach as it introduces the notions that are difficult to express in Agda, and presumably any other existing theorem prover.

APL is a language that pioneered the concept of rank- and shape-polymorphic programming. Expressions in APL are written in index-free combinator style with few syntactic rules. The language is dynamically typed, and each combinator is an operation on (multi-dimensional) arrays. Consider the following (valid) APL expression:

$$b ← 2 ÷̈ (1 ⌽ a) + ¯1 ⌽ a \qquad b_i = \frac{1}{2}\left(a_{(i-1)\%n} + a_{(i+1)\%n}\right)$$

It computes a two-point convolution of the array a using cyclic boundaries. This is done by first rotating vectors along the last axis of a one element to the left (¯1 ⌽ a), then one element to the right (1 ⌽ a), then adding these results element-wise (+), and then dividing each element by two (2 ÷̈). APL expressions such as this one are applicable to a of *any* rank, including zero-dimensional arrays. Not only has the initial set of APL combinators been found useful in practice, but it also gives rise to the number of universal equalities such as (-x) ⌽ x ⌽ a ≡ a, which says: if we first rotate vectors in the last axis of a by x elements in one direction and then rotate by x elements in the opposite direction, we will always get back the same array. These universal equalities are

---

[5]Here we show 1-dimensional versions of the functions, but in reality we implement rank-polymorphic cons/hd/tail in a way similar to sel from above.

based on simple arithmetic facts, yet they give a powerful reasoning technique and they can be used as rewrite rules for automatic program transformations.

5.1 Embedding of APL

The semantics of each APL operator is heavily overloaded: the same symbol has different meanings depending on how many arguments are being passed and what these arguments are, *i.e.* their shapes, sign, *etc.* For example, consider the `/` (slash) symbol that can be used as follows:

`+/a`	sum array elements, `+` is an argument
`2+/a`	sum in groups of 2, `+` and `2` are arguments
`2/a`	replicate each element 2 times
`+/[k]a`	sum over the k-th axis, `[k]` is an optional axis specification

While the embedding does not have to match the original syntax one-to-one, we would like to preserve one behaviour of the operators that is used incredibly often — the automatic cast between scalars, vectors, and multi-dimensional arrays. In APL every object is an array, therefore vectors and scalars can be simply used as arguments to the functions that expect arrays. Shapes of arrays are 1-dimensional arrays themselves. Replicating such a behaviour in Agda would lead to infinite recursion: we would have to index `Ar` type with `Ar`, which is not possible. Furthermore, binary operations in APL have the following casting behaviour:

`1 2 3 + 1`	computes to `2 3 4`
`1 + 1 2 3`	computes to `2 3 4`
`1 2 3 + 1 2 3`	computes to `2 4 6`
`1 2 3 4 + 1 2 3`	runtime error

If one of the arguments to the binary operation is a singleton array, it is automatically replicated to match the shape of the other element.

Normally, overloading in Agda is solved by using instance arguments. These are special kind of implicit arguments that are resolved using instance resolution, achieving a similar effect as classes and instances in Haskell. In our case, we define a relation dy-args between the ranks and shapes of the arguments of the binary operation:

```
data dy-args : ∀ m n → Vec ℕ m → Vec ℕ n → Set where
  n-n : ∀ {n s} → dy-args n n s s
  n-0 : ∀ {n s} → dy-args n 0 s []
  0-n : ∀ {n s} → dy-args 0 n [] s
```

The constructors of dy-args specify valid ways of calling a binary operation: either the shapes are identical, or one of them is a scalar (rank zero, shape empty). However, when we register these constructors as instances, Agda fails to resolve them when two zero-dimensional arrays are supplied as arguments. In this case all three instances fit, but Agda can only accept a unique solution. Ironically, in this case, all the three instances would lead to the same correct result.

We solve this problem by using metaprogramming: we define a macro and use it to resolve a given hidden argument. Within the macro, we are free to make arbitrary choices in case of non-unique solutions. Concretely, we define a macro dy-args-ok? that tries to construct an element of type dy-args m n sx sy. We then define a lifting function for binary operations as follows:

```
dy-args-dim : ∀ {m n sx sy} → dy-args m n sx sy → ℕ -- pick the largest rank
dy-args-shp : ∀ {m n sx sy} → (dy : dy-args m n sx sy) → Vec ℕ (dy-args-dim dy)
```

dy-type : ∀ a → Set a → Set a
dy-type a X = ∀ {m n sx sy} {@(tactic dy-args-ok?) args : dy-args m n sx sy}
　　　　　　　→ Ar X m sx → Ar X n sy → Ar X _ (dy-args-shp args)

lift′ : ∀ {a}{X : Set a} → (_⊕_ : X → X → X) → dy-type a X
lift′ _⊕_ {args = n-n} (imap a) (imap b) = imap λ iv → a iv ⊕ b iv
lift′ _⊕_ {args = n-0} (imap a) (imap b) = imap λ iv → a iv ⊕ b []
lift′ _⊕_ {args = 0-n} (imap a) (imap b) = imap λ iv → a [] ⊕ b iv

We define the dy-args-dim and dy-args-shp to pick the largest rank and shape from the arguments that are related by dy-args. The lift′ function itself turns any binary operation on array elements into a binary operation on arrays that replicates scalars correctly. Here we demonstrate the lifting _+_ for natural numbers.

_+_ = lift′ $\mathbb{N}$._+_	ex_1 ex_2 : Ar $\mathbb{N}$ 3 s		ex_4 ex_5 ex_6 : ∀ {n s}	ex_4 x = x + x
a : Ar $\mathbb{N}$ 3 s	ex_1 = a + z	ex_3 : Ar $\mathbb{N}$ 0 []	→ Ar $\mathbb{N}$ n s	ex_5 x = x + z
z : Ar $\mathbb{N}$ 0 []	ex_2 = a + a	ex_3 = z + z	→ Ar $\mathbb{N}$ n s	ex_6 x = z + x

In this example, a is a 3-d array, and z is a scalar. The lifted addition on arrays admits all the desired variants. The last three examples on the right show that it still works for the cases when the rank is not known statically.

We have only presented the overloading between the Ar types of different shapes. This still does not solve the problem of implicit casts from base types such as $\mathbb{N}$ and vectors into arrays. However, this can be solved by defining regular instances. In the code accompanying this paper, we define a similar lift function that extends the domain of the lifted binary operation and accepts base types, vectors and arrays, and their combinations.

5.2 A Convolutional Neural Network

As a practical application, we consider a convolutional neural network for recognising hand-written digits, implemented in APL. The reference implementation we start from [Šinkarovs et al. 2019] is written entirely in APL without relying on any external libraries or frameworks. The implementation is very concise — apart from built-in operators, it only defines 10 new functions, each of which is a single line of APL code. Translating these functions into our embedded array language serves two purposes. First, we stress-test abstractions used in our embedding and the extractor capabilities. Second, we verify that all the shapes and ranks match, the indexing is in-bound, no division by zero occurs, and that the functions are terminating. As APL is dynamically typed, it is difficult to be sure that no runtime errors will occur. Embedding the code into Agda essentially requires us to define a type system for the operators in use and guarantee that they hold. We consider three representative samples of our encoding and explain the details.

Logistic function. After the convolution and fully-connected layers in the CNN, the activation function is applied to each of the results. The activation function in use is called the standard logistic function $\frac{1}{1-e^x}$, and it is being applied to all the elements of the resulting array. Here is the implementation in APL and in our embedding:

```
logistic←{÷1+*-ω}
```

logistic : ∀ {n s} → Ar Float n s → Ar Float n s
logistic ω = ÷$_r$ 1.0 +$_r$ *$_r$ -$_r$ ω

As can be seen, the implementations are almost identical. There are two important reasons: the ability to define the precedence and the associativity of the operators; and the automatic casts

that we explained before. All the operators in APL are right-associative, which we implement in Agda using infixr statements. We distinguish the operations on base types by adding a postfix to the name, so instead of `_+_`, `_-_`, *etc.* we have `_+ᵣ_`, `_-ᵣ_` when we the arguments are arrays of base type Float. If we read the body right to left, the function negates (`-ᵣ_`) its argument, then it computes the exponent (`*ᵣ_`) of that result, then it adds 1.0 to all the elements, and finally it takes a reciprocal (`÷ᵣ_`). The function is shape- and rank-polymorphic; it does not require additional proofs and it normalises to a single imap.

Mean Squared Error. The nice behaviour of the above function is not really surprising since it just maps scalar operations over individual array elements. However, this is a common pattern in array-based applications. Here is another example that is used to compute the mean error which is a sum of squared elements divided by two:

```
meansqerr←{÷∘2+/,(α-ω)*2}
```
```
meansqerr : ∀ {n s} → Ar Float n s → Ar Float n s → Scal Float
meansqerr α ω = _÷ᵣ 2.0 $ _+ᵣ′_ / , (α -ᵣ ω) ×ᵣ (α -ᵣ ω)
```

In addition to element-wise mapping we have a reduction of the elements — the `_/_` operator. On the right hand side it gets an array that is being reduced, and the left operator is a binary function that performs the actual operation. We have a flattened (`,_`) square of differences on the right, and addition on Floats on the left. We need to flatten the array on the right because according to the APL semantics, `_/_` reduces over the last axis of the array. Also, in contrast to reductions found in many functional languages, APL does not require the default element but deduces it from the operation in use. We have encoded the same behaviour using instance resolution. However, we had to supply the addition on floats `_+ᵣ′_`, rather than our generalised addition on the arrays and vectors of floats `_+ᵣ_`, because otherwise Agda fails to instantiate hidden arguments to `_+ᵣ_`. Finally, partial application of division on the right `÷∘2` is a built-in feature of mixfix operators in Agda.

Back Average Pool. The reverse average pooling function requires us to specify a shape restriction: the shape of the result must be twice as big as the shape of the input array (in every dimension).

```
backavgpool←{2⊢2/ω÷4}⍨2
```
```
backavgpool : ∀ {s} → Ar Float 2 s → Ar Float 2 $ ▼ (2 × s)
backavgpool {s = _ ∷ _ ∷ []} ω = 2 ⊬ᵣ 2 /ᵣ′ ω ÷ᵣ 4.0
  where
    infixr 20 _/ᵣ′_
    _/ᵣ′_ = _/ᵣ_ {s = _ ∷ []}
```

We specify this relation using our lifted arithmetic operations: 2 × s, where the left argument is of type ℕ, and the right argument is Vec ℕ 2. The multiplication returns a 1-dimensional array of type Ar ℕ 1 (2 ∷ []), and we typecast it back to Vec using the ▼_ function.

The function itself divides all the array elements by 4.0 and replicates them two times across each row (`_/ᵣ_`), and two times across each column (`_⊬ᵣ_`). Note that we have to help Agda by specifying that s is guaranteed to be of length 2. Also, similarly to before, we need to supply a hidden argument to `_/ᵣ_`. Rather than doing this inside the application chain, we used a where syntax to define a local variant of the row replicator `_/ᵣ′_`.

Average Pooling. Our final example is an average pooling function. It takes a two-dimensional array of floats as an argument, where each axis is divisible by two. It partitions the array into sub-arrays of shape [2,2] and computes the average of each partition. Here is the implementation:
```
avg ← { (+/÷≢),ω}
```

```
avgpool ← { (x y) ← ρω ⋄ avg�2 ⊢ 0 2 1 3⍉(x÷2) 2 (y÷2) 2ρ ω }
```

avgpool : ∀ {s} → Ar Float 2 $ ▼ (s × 2) → Ar Float 2 s
avgpool {s} (imap p) = imap $ λ iv →
  let ix , ix<s = ix→a iv
      q   = λ (i , pf) → p $ a→ix ((ix × 2) + i) (s × 2) (A<B⟹K<2⟹A*2+K<B*2 ix<s pf)
      [2,2]   = cst {s = 2 ∷ []} 2
  in ▼ (_÷ᵣ 4.0 $ _+ᵣ′_ / , q ¨ ι [2,2])

In this example, a direct implementation that uses indexing is actually more straightforward than one expressed in index-free style. The result of average pooling is given by the imap. Reading the body of the imap right to left, we obtain an array of indices (ι_) into a two-dimensional array of shape [2,2]. Then for each element (_¨_) in that array we apply the function f bound above it. Then we sum the elements up and divide them by 4.0. The indices returned by (ι_) are dependent pairs where the first component is a 1-dimensional array representing the value of the index, and the second component is a proof that the index is strictly less than the array shape (in our case [2,2]). In q, we pattern-match on the pair, and we compute selection into the argument of avgpool at index $2iv + i$. The final argument to q is a proof that this index is within the bounds of the array.

Here we consider the extraction of avgpool into SAC, slightly reformatted for better readability.

```
float[.,.] avgpool(int[2] x_1, float[.,.] x_3) {
  float[.,.] __ret;
  s = x_1;
  assert (shape (x_1)[0] == 2);
  assert (take (2, shape (x_3))
           == cons ((x_1[0] $* 2), cons ((x_1[1+0] $* 2), empty ([]))));
  #define p(__x) (x_3)[__x]
  __ret = with {
    (.<= iv_1 <=.) {
       i = iv_1[0];
       j = iv_1[1+0];
    } : (   (p(cons(((i $* 2) $+ 0), cons(((j $* 2) $+ 0), []))))
         $+ (p(cons(((i $* 2) $+ 0), cons(((j $* 2) $+ 1), []))))
         $+ (p(cons(((i $* 2) $+ 1), cons(((j $* 2) $+ 0), []))))
         $+ (p(cons(((i $* 2) $+ 1), cons(((j $* 2) $+ 1), []))))
         $+ 0.0f))))
        $/ 4.0f);
  }: genarray (s, zero_float ([]));
  assert (take (2, shape (__ret)) == x_1);
  return __ret;
}
```

In the extracted code, all the local definitions are inlined, as well as all the compound array operations. We are very close to the code that a programmer could write. The assertions at the top are deduced from the type signature: the first argument must be a two-element array, and the shape of the second argument is twice the shape of the first argument. We use arithmetic operations prefixed with $, to indicate that these are operations on scalars (int and float) to help the compiler

with instantiating overloadings. Before returning, the function asserts that the shape of the returned result must be equal to the first argument. The body of the with-loop performs 4 selections into the argument array and averages them. Finally, since SaC is a first-order language but imap is a higher-order construct, the extractor has inserted a macro p to mimic the application of the pattern-matched argument of imap as a function.

6 RELATED WORK

Metaprogramming. There is a large body of work on metaprogramming facilities in various programming languages. Herzeel et al. [2008] track the origins of metaprogramming to Smith's work on reflection in Lisp [Smith 1984]. Some prominent metaprogramming systems include MetaOcaml [Kiselyov 2014], MetaML [Taha and Sheard 1997], reFlect [Grundy et al. 2006], Template Haskell [Sheard and Peyton Jones 2002], Racket [Flatt and PLT 2010], and various other Lisp/Scheme dialects. However, these systems typically do not support dependent types, so they are not well suited for our goal of statically enforcing correctness of embedded programs.

Embedding. Defining deep embeddings with static guarantees are a common application of dependent types [Allais et al. 2018; Altenkirch and Reus 1999; Chapman 2009; Danielsson 2007; McBride 2010]. These embeddings usually also define semantics of the embedded language and therefore allow us to reason about the correctness of program transformations and optimisations. While the fact that this is possible is impressive in theory, the resulting encodings are often difficult to use in practice. In this paper we instead aim for a more lightweight approach.

Svenningsson and Axelsson [2013] propose to solve this problem with a combination of deep and shallow embeddings. Their idea is to define a small deep embedding and leverage type classes in Haskell to define the rest of the language on top of that. It would be interesting to see whether such an approach scales to dependently-typed embedded languages.

Extraction. The Coq proof assistant is equipped with extraction capabilities [Letouzey 2003, 2008], which extracts functional code from Coq proofs (or programs). The default target language is Ocaml, but a few other options were added recently. Likewise, Agda itself has a mechanism for defining custom backends, of which the GHC backend is the most prominent. Other proof assistants provide similar extraction tools as well. The main difference from our approach in this paper is that these extractors are written as plugins to the proof assistant, while we implement our extractors directly in the proof assistant itself. While it would be possible to implement extractors presented in this paper as Coq plugins or Agda backends, conceptually they are more heavyweight. Our extractors and programs can (in principle) communicate with each other. In addition, as they are just Agda programs, they can be reflected themselves and their structure can be leveraged.

Dependently typed metaprogramming. Several dependently-typed languages are equipped with metaprogramming capabilities: Idris [Christiansen and Brady 2016], Lean [Ebner et al. 2017], Coq [Anand et al. 2018], and Agda [van der Walt and Swierstra 2013]. All of these implement a similar API as described in this paper. This is reassuring, as it means our proposed approach is immediately portable into many other contexts. Chang et al. [2019] introduce the Turnstile+ framework for building dependently typed embedded DSLs and very much shares the ideas advocated in this paper, suggesting that our approach could work there as well. Sozeau et al. [2019] use MetaCoq to formally verify the core type system of Coq. This combines nicely with our approach, as we could use the verified core language as a basis to verify our custom extractors. Annenkov et al. [2020] use MetaCoq to implement a DSL combining deep and shallow approaches, in a way that is quite similar to our own. While they are able to formally reason about preservation of semantics (which

we can't do yet), it is unclear whether their approach scales to dependently-typed embedded languages.

Arrays. Using dependent types to verify properties of array programs is not a novel idea. For example, Qube [Trojahner and Grelck 2009] and Remora [Slepak et al. 2014] are dependently typed languages that are focused on array programming. Both of these focus on automating the type inference, which is a big advantage for programmers. However, one has to rely on the capabilities of the inference engine, which may fail for some complex examples. In [Thiemann and Chakravarty 2013] authors use Agda as a frontend for Accelerate, an array library in Haskell. The motivation of the work is similar to ours, to provide static guarantees about array computations. As the target language of this work is Haskell, and Agda provides a backend for it, the integration happens smoothly without requiring any extraction techniques.

7 CONCLUSIONS AND FUTURE WORK

In this paper we investigate the idea of developing embedded programs hand-in-hand with custom code generators for them. We solve the well-known conundrum of choosing between deep and shallow embedding by leveraging the power of reflection. This allows us to enjoying the benefits of shallow embedding, while keeping full access to the internal structure of the embedded programs.

We apply this idea in the context of dependently-typed languages to create verified implementations that can be used in the context of an existing programming language. We embed the target language in a theorem-prover, using dependent types to encode the properties of interest. Using reflection we bring the verified implementation back into the original language.

We have demonstrated the approach by implementing three embedded languages and two extractors, using Agda as our host. Along the way we made some improvements to the reflection API of Agda, and in the end we used our embedding to implement (and verify) a practical application — a convolutional neural network.

The main advantages of our approach are twofold. First, our solution is fine-grained — we can chose what part of the application to embed, and what constructs of the host language to extract. Secondly, our extractors can use the full power of dependent types to guarantee safety of our embedded programs.

Right now we cannot yet guarantee that the extracted code preservers the semantics of the original implementation. While we rarely see fully-verified compiler backends in the real world, our approach is very close to enabling this. We would need a formal semantics of the reflected language and the proof that reflected programs respect it. While this is non-trivial, a system like Agda could do this in principle.

There is a number of improvements that can be added to Agda and our extractors to make the resulting code more efficient. Supporting lets in the internal syntax would help to preserve sharing. Recognising irrelevance annotations in the extractors would help to eliminate unused function arguments. Introducing proper language primitives to specify what exactly is an embedding would be helpful. And finally, having access to more of Agda's internals such as case trees would help to generate more performant code.

Overall, this work only scratches the surface of extraction-based compilation. We never considered alternative theories supported by theorem provers, *e.g.* cubical type theory in Agda; we did not consider recursive metaprogramming; we did not consider integrating optimisations of extracted programs other than what rewriting rules are capable to do. All of these offer exciting research opportunities on the way to make verified software easily accessible. To paraphrase Jim Morrison: "Safety and no surprise, The End".

REFERENCES

Agda development team. 2021. Agda v2.6.1.3 Documentation. https://agda.readthedocs.io. Accessed [2021/02/20].

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Proc. ACM Program. Lang.* 2, ICFP, Article 90 (July 2018), 30 pages. https://doi.org/10.1145/3236785

Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S095679681500009X

Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic (CSL '99)*. Springer-Verlag, Berlin, Heidelberg, 453–468.

Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 20–39.

Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: A Smart Contract Certification Framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 215–228. https://doi.org/10.1145/3372885.3373829

Eli Barzilay. 2005. *Implementing Reflection in Nuprl.* Ph.D. Dissertation. USA. Advisor(s) Constable, Robert. AAI3195788.

Yves Bertot and Pierre Castran. 2010. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions* (1st ed.). Springer Publishing Company, Incorporated.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23 (9 2013), 552–593. Issue 05. https://doi.org/10.1017/S095679681300018X

Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. 2019. Dependent Type Systems as Macros. *Proc. ACM Program. Lang.* 4, POPL, Article 3 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371071

James Chapman. 2009. Type Theory Should Eat Itself. *Electronic Notes in Theoretical Computer Science* 228 (2009), 21–36. https://doi.org/10.1016/j.entcs.2008.12.114 Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).

David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 284–297. https://doi.org/10.1145/2951913.2951932

Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. 2021. The Taming of the Rew: A Type Theory with Computational Assumptions. *Proc. ACM Program. Lang.* 5, POPL, Article 60 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434341

Nils Anders Danielsson. 2007. A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family. In *Types for Proofs and Programs*, Thorsten Altenkirch and Conor McBride (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 93–109.

Nils Anders Danielsson and Ulf Norell. 2011. Parsing Mixfix Operators. In *Implementation and Application of Functional Languages*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–99.

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description).. In *CADE (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. http://dblp.uni-trier.de/db/conf/cade/cade2015.html#MouraKADR15

Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 34 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110278

Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

C. Grelck. 2012. Single Assignment C (sac): High Productivity Meets High Performance. In *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary (Lecture Notes in Computer Science, Vol. 7241)*, V. Zsók, Z. Horváth, and R. Plasmeijer (Eds.). Springer, 207–278. https://doi.org/10.1007/978-3-642-32096-5_5

Clemens Grelck and Sven-Bodo Scholz. 2006. Sac: A Functional Array Language for Efficient Multithreaded Execution. *International Journal of Parallel Programming* 34, 4 (2006), 383–427. https://doi.org/10.1007/s10766-006-0018-x

Jim Grundy, Thomas F. Melham, and John W. O'Leary. 2006. A reflective functional language for hardware design and theorem proving. *J. Funct. Program.* 16, 2 (2006), 157–196. https://doi.org/10.1017/S0956796805005757

Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt. 2008. Reflection for the Masses. In *Self-Sustaining Systems*, Robert Hirschfeld and Kim Rose (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 87–122.

Kenneth E. Iverson. 1962. *A Programming Language.* John Wiley & Sons, Inc., New York, NY, USA.

Oleg Kiselyov. 2014. The Design and Implementation of MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

Pierre Letouzey. 2003. A New Extraction for Coq. In *Types for Proofs and Programs*, Herman Geuvers and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 200–219.

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 359–369.

LLVM development team. 2021. My First Language Frontend with LLVM Tutorial. https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html. Accessed [2021/02/20].

Per Martin-Löf. 1998. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*, Giovanni Sambin and Jan M. Smith (Eds.). Oxford Logic Guides, Vol. 36. Oxford University Press, 127–172.

Conor McBride. 2010. Outrageous but Meaningful Coincidences: Dependent Type-Safe Syntax and Evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming* (Baltimore, Maryland, USA) *(WGP '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1863495.1863497

Conor McBride and James McKinna. 2004. The View from the Left. *J. Funct. Program.* 14, 1 (Jan. 2004), 69–111. https://doi.org/10.1017/S0956796803004829

Lenore M. Restifo Mullin. 1988. *A Mathematics of Arrays*. Ph.D. Dissertation. Syracuse University.

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming* (Heijen, The Netherlands) *(AFP'08)*. Springer-Verlag, Berlin, Heidelberg, 230–266.

Sven-Bodo Scholz. 2003. Single Assignment C: Efficient Support for High-Level Array Operations in a Functional Setting. *J. Funct. Program.* 13, 6 (Nov. 2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 Haskell Workshop, Pittsburgh* (proceedings of the 2002 haskell workshop, pittsburgh ed.). 1–16.

Artjoms Šinkarovs, Sven-Bodo Scholz, Robert Bernecky, Roeland Douma, and Clemens Grelck. 2014. SaC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. *Concurrency and Computation: Practice and Experience* 26, 4 (2014), 952–971. https://doi.org/10.1002/cpe.3078 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3078

Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–46.

Brian Cantwell Smith. 1984. Reflection and Semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Salt Lake City, Utah, USA) *(POPL '84)*. Association for Computing Machinery, New York, NY, USA, 23–35. https://doi.org/10.1145/800017.800513

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 8 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371076

Josef Svenningsson and Emil Axelsson. 2013. Combining Deep and Shallow Embedding for EDSL. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–36.

Bo Joel Svensson and Josef Svenningsson. 2014. Defunctionalizing Push Arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing* (Gothenburg, Sweden) *(FHPC '14)*. ACM, New York, NY, USA, 43–52. https://doi.org/10.1145/2636228.2636231

Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. *SIGPLAN Not.* 32, 12 (Dec. 1997), 203–217. https://doi.org/10.1145/258994.259019

Peter Thiemann and Manuel M. T. Chakravarty. 2013. Agda Meets Accelerate. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 174–189.

Kai Trojahner and Clemens Grelck. 2009. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming* 78, 7 (2009), 643–664. https://doi.org/10.1016/j.jlap.2009.03.002 The 19th Nordic Workshop on Programming Theory (NWPT 2007).

Paul van der Walt and Wouter Swierstra. 2013. Engineering Proof by Reflection in Agda. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

Artjoms Šinkarovs, Robert Bernecky, and Sven-Bodo Scholz. 2019. Convolutional Neural Networks in APL. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Phoenix, AZ, USA) *(ARRAY 2019)*. Association for Computing Machinery, New York, NY, USA, 69–79. https://doi.org/10.1145/3315454.3329960